v1.2
20140902

Programmazione Avanzata e Paradigmi

Ingegneria e Scienze Informatiche - UNIBO
a.a 2013/2014
Lecturer: Alessandro Ricci

[module 1.2]
FUNCTIONAL PROGRAMMING INTRODUCTION

SUMMARY

- Functional Programming - main ingredients
  - functions and expressions
    - abstraction, applications
  - functions as expressible values
    - lambda expressions
  - computation as reduction
  - evaluation strategies
    - call-by-value, call-by-name, lazy evaluation

THE FUNCTIONAL PARADIGM

- (Mathematical) functions as model of computation is the function
  - function are treated as first-class object
  - can be recursive, high-order, polymorphic
- Computation is carried out entirely through *evaluation of expressions*
  - evaluating expressions = rewriting (reducing) expressions
- No state modification (like in the imperative)
  - variables are immutable bindings to symbols
    - not memory location
  - no assignment
  - no iterations and loops
    - recursion is used instead

FUNCTIONS AND EXPRESSIONS

- Functional languages have no commands, only expressions
- Two main constructs for defining expression (a part of primitive data values and operators)
  - **abstraction** (λx.<exp>)
    - given an expression <exp> and an identifier x allows the construction of an expression (λx.<exp>) denoting a function that transforms the formal parameter x into <exp>
      - <exp> is "abstracted" from the specific value bound to x
  - **application** (<f_exp> <a_exp>)
    - the application of an expression <f_exp> to another <a_exp>, which denotes the application of a function denoted by <f_exp> to the argument denoted by <a_exp>

EXAMPLE - HASKELL

- Function definition. Syntax:

```
<func name> :: <func type>
<func name> <formal params> = <expression>

myFunc :: Int -> Int –> Int
myFunc x y = x + y + 1
```

- Evaluating the expression using an Haskell REPL

```
> myFunc 1 2
4
```

- Definitions can be used in general for binding values to symbols:

```
aValue :: Int
aValue = 5
```

EXAMPLE - HASKELL

- Function application. Syntax:

  ```
  <func name> <actual param expr>
  ```

- Example - definition of a new function with function application in the body:

  ```
  myFunc2 :: Int -> Int -> Int -> Int
  myFunc2 x y z = z * myFunc x y
  ```

FUNCTION AS EXPRESSIBLE VALUES

- Functions can be treated as *values*
  - specific syntax for an expression which denotes a function (value)
    - i.e. it is possible to write a function without having necessarily to assign it a name
  - they have a type
- Outcome
  - can be assigned to variables, can be passed as a parameter, can be returned as the result of a function...
    - *high-order functions* = functions that have other functions as parameter

LAMBDA EXPRESSIONS IN HASKELL

- Expressions denoting functional values. Syntax:

```
\ <params> -> <body>
```

- Can be used in every place where expressions/values of the specified type (which is functional type)

```
> (\x -> x+1) 3
4

fun :: (Int -> Int -> Int) -> Int -> Int
fun f a = f a a

> fun (\x y -> x + y) 5
10
```

IN SCALA

- Example of lambda expression in Scala (using Scala REPL):

```
scala> val increase = (x: Int) => x + 1
increase: (Int) => Int = <function1>

scala> increase(10)
res: Int = 11
```

- Actually Scala allows to use also statements in function bodies (being Scala also imperative OOP):

```
val myStrangeFunc = (x: Int) => {
  println(“how")
  println("are")
  println("you")
   x + 1
}
```

THE TYPES OF FUNCTIONS

- The type of a function is expressed in terms of the type of the arguments and of the result
- Example in Haskell:
  - given a function

    `f <p1> <p2> ... <pn> = <expr>`

    where <t1> is the type of <p1>,...,<tn> is the type of <pn> and <tres> is the type of the result of the evaluation of <expr>, then the type of the function is denoted as:

    `<t1> -> ... -> <tn> -> <tres>`
  - example: the type of a function add summing 2 integers

    `add x y = x + y`

    can be denoted as: `Int -> Int -> Int`

    where Int is the integer primitive type

TYPE EXPRESSION

- Explicit declaration of the type of a function (when defining the function)

```
succ :: Int -> Int
succ n = n + 1
```

- Haskell's static type system defines the formal relationships between types and values
  - ensuring that Haskell program are type safe, i.e. it does not allow to mismatch the types
- If omitted, it is *inferred* by the system
  - type inference, discussed in next module

ONLY ONE PARAMETER IS ENOUGH: CURRYING

- High-order functions make it possible to express any *N* params function as a 1-param function
  - whose return param is a further function
- Example: the function with 2 parameters:

  `(λ x y . x + y)`

  can be expressed as:

  `(λ x . (λ y . x + y))`
  - the second function is 1-param function returning a 1-param function
- The transformation of a N-params function into a chain of 1-param function is called **currying**
  - "the function is curried"

CURRYING IN HASKELL

- In Haskell every function has actually just one parameter
  - the function definition `f x y z = <body>`
    is just syntactic sugar for: `f x = \y -> (\z -> …)`
  - example

```
add x = \y -> x + y
succ = add 1

> succ 4
5
```

CONDITIONAL EXPRESSIONS

- *if* control structure is modeled as a *conditional* expression
  - it denotes a value, which depends on the value of the predicate, rather then a sequence of statements
  - vs. command in imperative languages
- In Haskell:

```
if <expr>
   then <expr>
   else <expr>
```

  - example:

```
max x y z = if (x > y) && (y > z)
               then x
               else if (y > z)
                       then y
                       else z
```

GUARDS

- In Haskell *guards* can be used instead of conditional expressions to have a more declarative style in defining functions. Syntax:

```
<fname> <formal params>
        | <guard1> = <exp1>
        | <guard2> = <exp2>
        …
        | otherwise = <expn>
```

The meaning is: the definition of the function <fname> is <exp1> if the conditions <guard1> holds, otherwise is <exp2> is <guard2> holds,… If no conditions (guards) hold, then the definition is <expn>

where guards <guardn> are boolean expressions expressing some condition on parameters

  - `otherwise` is not compulsory

- Example:

```
max x y z
      | x > y && x > z     = x
      | y > z              = y
      | otherwise          = z
```

NO ITERATION, ONLY RECURSION

- In stateless computational models, loops and iterations *disappear* and **recursion** becomes the fundamental construct for sequence control
  - iteration looses sense without assignment
- Examples of recursion in Haskell:

```
fact n | n <= 1 = 1
       | otherwise = n*fact(n - 1)
```

DECLARATIVE STYLE

- Expressions don't express *sequences* of commands to do
  - they are more a description of what to compute, not how to compute it
  - there could be different evaluation strategy expressing how to reduce the expression
- Benefits
  - no side effects
  - *referential transparency*

REFERENTIAL TRANSPARENCY

- Pure declarative style => referential transparency property
  = languages are referentially transparent
- Meaning:
  - "equals can be replaced by equals", no side effects
- Example
  - Given an expression

    ...x+x...

    where x = f a
  - the function application (f a) may be substituted for any free occurrence of x in the scope created by the where expression
    - e.g. in the x+x expression
- Referential transparency enables the possibility of doing equational reasoning
  - reasoning formally about programs and their properties
  - informally in writing and debugging programs

DECLARATIVE STYLE LIMITS

- Not every programming problem/aspect can be effectively modeled as a function
- Some side effects are unavoidable and wanted
  - *I/O* interactions
    - e.g. how to model a print command?
  - keeping track and updating some *state*
  - modeling actions & sequences of actions
  - keeping track of time
  - runtime errors
  - ...
- Mechanisms extending pure FP
  - monads (next module)

ERRORS

- The possibility to handle situation at runtime in which some kind of errors occur is an important aspect of any programming language
  - e.g. dividing a number by zero, getting an element from an empty list, etc
  - e.g. exception handling in modern OOP languages
- In the theory, these situations may correspond to cases in which a function cannot be evaluated or whose evaluation is undefined
  - or, alternatively, we may model a function to include the error values in its co-domain
    - not an ideal solution

ERRORS IN HASKELL

- Built-in `error:: String -> a` function
  - stops the program, printing a message
  - example:

```
tail :: [a] -> a
tail (_:xs) = xs
tail [] = error “Error: attempting to get the \
                  tail of an empty list!”
```

  - note: la definizione in questo caso usa il meccanismo di pattern matching - che si vedrà nel prossimo modulo. Il meccanismo permette di definire una funzione mediante più definizioni, ognuna delle quali vale quando i parametri passati soddisfano un certo pattern

SEMANTICS OF COMPUTATION: EVALUATION

COMPUTATION AS REDUCTION

- **Evaluation** = the procedure used to transform a complex expression into its value
- In FP evaluation is based simply on expression **reduction,** which consists in *rewriting* an expression replacing sub-expressions with (simpler) expressions
  - in a complex expression, a sub expression of the form (f x) is textually replaced by the body of the function in which formal parameters are replaced by the actual parameters

COMPUTATION AS REDUCTION: EXAMPLE

- Being the recursive function:
  `fact n = if (n <= 1) then 1 else n*fact(n - 1)`
  let's compute `fact 3`
- ```
  fact 3
  -> (if (n <= 1) then 1 else n*fact(n - 1)) 3
  -> if (3 <= 0) then 1 else 3*fact(3 - 1)
  -> 3*fact(3 - 1)
  -> 3*fact 2
  -> 3*((if (n <= 1) then 1 else n*fact(n - 1)) 2)
  -> 3*(if (2 <= 1) then 1 else 2*fact(1))
  -> 3*(2*fact(1))
  -> 3*(2*((if (n <= 1) then n else n*fact(n - 1)) 1)
  -> 3*(2*(if (1 <= 1) then 1 else 1*fact(1 - 1))
  -> 3*(2*(1))
  -> 6
  ```
  ```

DIVERGING COMPUTATIONS

- In some cases rewriting is not going to converge
- For instance, let's consider:

```
f x = f (f x)
```

- Evaluating: `f 1`

```
f 1
-> ((f (f x)) 1)
-> (f (f 1))
-> (f ((f (f x)) 1))
-> (f (f (f 1)))
-> …
```

REDUCTION - DEFINITIONS

- **Redex** (Red-ucible ex-pression)
  - a redex is an application of the form (<f_exp> <a_exp>), where <f_exp> is the expression of a function, either anonymous (λx.<body>) or not
- **Reductum**
  - the reductum of a redex ((λx.<body>) <a_exp>) is the expression which is obtained by replacing in <body> each occurrence of the formal parameter <a_exp> by a copy of <a_exp>, *avoiding variable capture*
- **β-rule**
  - an expression <exp>, in which a redex appears as a subexpression is reduced (or rewrites, simplifies) to <exp1> (notation <exp> -> <exp1>), where <exp1> is obtained from <exp> by replacing the redex by its reductum.

CAPTURE FREE SUBSTITUTIONS

- In an abstraction expression (λx.<body>), the parameter can be renamed and the meaning of the expression does not change
  - e.g. (λxy.x+y) is equivalent to (λ*a*b.*a*+*b*)
- Renaming is necessary when the parameter passed to the function is the same of a *free* variable inside the body
  - free variable = variable not being bound by some λ as a parameter
  - e.g. ((λx.λy.x+y) y)
    - y is a free var in the outer expression and the name of a parameter in the abstraction
    - if we don't rename the parameter inside the inner expression (λy.x+y), the free variable is "captured", reducing to (λy.y +y) which is wrong
- In doing in the evaluation, renaming is performed to avoid capturing
  - ((λx.λy.x+y) y) ≡ ((λx.λw.x+w) y) -> λw.y+w

EVALUATION

- A program is a series of value definitions
  - each of which inserts a new association into the environment
  - can require the evaluation of arbitrarily complex expression
- The semantics of computation is operationally given by the **evaluation** process
  - symbolic rewriting of strings (reduction), repeatedly using 2 main operations to simplify expressions until they reach a simple form which immediately denotes a value
    1. simple search of an identifier through the environment
    2. when an identifier is determined as being bound in an environment, replace the identifier by its definition
- Termination
  - the evaluation process proceeds until the expression is a *value*
  - values are expressions which cannot be further rewritten
    - values of primitive type, functional values

REMARK

- Every expression of the form (λx.<exp>) represents directly a value, so redexes possibly contained in <exp> are never rewritten until the expression is applied to some argument
  - in other words, in functional languages evaluation does not occur under abstractions
- So for instance the result of the evaluation of the expression:

  (λx.((λy.y+1) 2))

  is *not* the primitive value 3, but the expression itself:

  (λx.((λy.y+1) 2))

EVALUATION STRATEGIES

- An expression can have multiple redexes.
- For example, given the function definitions:

```
K x y = x
r z = r(r(z))
D u = if (u = 0)
          then 1
          else u
succ v = v + 1
```

  then, what is result of the evaluation: `K (D (succ 0)) (r 2)` ?

- 4 possible redexes...
  - `K (D (succ 0))`
  - `D (succ 0)`
  - `succ 0`
  - `r 2`

EVALUATION STRATEGIES

- Two main approaches
  - **applicative order**
    - also called *call by value*
  - **normal order**
    - also called *call by name*
  - **lazy evaluation**
    - also called *call by need*
    - variant of the call by name

APPLICATIVE ORDER / CALL BY VALUE

- Also called *eager* evaluation or *strict* evaluation
- The leftmost, innermost redex is evaluated first
  - that is: a redex is evaluated only if the expression which constitutes its argument part is already a value
- Procedure:
  - scan the expr to be evaluated from the left, choosing the first application encountered. Let it be `(f_exp a_exp)`
  - first evaluate (recursively applying this method) f_exp until it has been reduced to a value (of a functional type) of the form `(λx. <body>)`
  - evaluate the argument part, a_exp, of the application, so that it is reduced to a value val
  - finally reduce the redex `(( λx. <body>) val)` and repeat from 1)

APPLICATIVE ORDER / CALL BY VALUE: EXAMPLE

- evaluation of: (K (D (succ 0))) (r 2)

```
(K (D (succ 0))) (r 2)
-> (K (D ((λv.v + 1) 0))) (r 2)
-> (K (D 1)) (r 2)
-> (K ((λu.if (u = 0) then 1 else u) 1)) (r 2)
-> (K 1)(r 2)
-> ((λx.λy.x) 1)(r 2)
-> (λy.1) (r 2)
-> (λy.1) (r (r 2))
-> (λy.1) (r (r (r 2)))
..
```

- the computation is diverging

APPLICATIVE ORDER / CALL BY VALUE

- Adopted by Lisp, Scheme, ML
  - not pure, side effects possible

NORMAL ORDER / CALL-BY-NAME

- Also called *non-strict* evaluation
- The leftmost, *outermost* redex is evaluated first
  - that is: a redex is evaluated before its argument part
- Procedure
  - scan the expr to be evaluated from left , choosing the first application. Let it be (<f_exp> <a_exp>)
  - first evaluate f_exp until it has been reduced to a value (λx.<body>)
  - reduce the redex ((λx.<exp>) <a_exp>) using the beta-rule and repeat the procedure

NORMAL ORDER / CALL-BY-NAME: EXAMPLE

- evaluation of: `(K (D (succ 0))) (r 2)`

```
(K (D (succ 0))) (r 2)
-> (λy.D (succ 0)) (r 2)
-> D (succ 0)
-> if (succ 0) = 0 then 1 else (succ 0)
-> if (1 = 0) then 1 else (succ 0)
-> succ 0
-> 1
```

- the result of the evaluation in this case is the value 1

LAZY EVALUATION

- Strategy also called ***call by need***
- Variant of the evaluation by name that avoid to reduce a redex multiple times
  - the first time a redex is evaluated, the result is propagated to every point of the expression
  - not string rewriting, but *graph* rewriting
  - adopted by all modern pure FP languages, such as Haskell, Miranda
- The example:

  ...

```
-> if (succ 0) = 0 then 1 else (succ 0)
-> if (1 = 0) then 1 else 1
-> 1
```

THEOREM

- Can different strategies produce distinct values for the same expression? For pure functional programming the answer is given by the following fundamental theorem:

> Let <exp> be a closed expression. If <exp> reduces to a primitive value <val> using any of the three strategy, then <exp> reduces to <val> following the **by-name** (normal order) strategy. If <exp> diverges using the by-name strategy, then it diverges also in the other 2 strategies.
>
> • closed expressions are expressions with all variables are bound.
> • primitive values do not include functional values

- Very important property which holds only if we consider pure FP, without side effects
  - property fundamental for reasoning about programs

EVALUATION STRATEGIES IN MODERN FP LANGUAGES

- Modern pure FP languages adopt lazy evaluation (call-by-name, normal-order)
  - examples: Haskell, Miranda
- FP Languages that allow for side effects (e.g. changing the state of a var) typically adopt call-by-value
  - examples: Lisp, Scheme, ML

BIBLIOGRAPHY

- Gabbrielli and Martini. *"Programming Languages: Principles and Paradigms".* Chapter 11
- Paul Hudak. *"Concepts, Evolution, and Application of Functional Programming Languages"*. ACM Computing Surveys, Vol. 21, No. 3, Sept 1989
- Paul Hudak, Joseph H. Fasel, *"A Gentle Introduction to Haskell"*. ACM SIGPLAN, Vol 27, No. 5, May 1992
- Thompson. "The Craft of Functional Programming"